Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 9 (2012) 116 – 125

Procedia Computer Science

International Conference on Computational Science, ICCS 2012

Headphone-based spatial sound with a GPU accelerator

Jose A. Belloch[a,1], Miguel Ferrer[b], Alberto Gonzalez[b], F. J. Martinez-Zaldivar[b], Antonio M. Vidal[c]

[a]*Instituto de las Telecomunicaciones y Aplicaciones Multimedia, Universitat Politecnica de Valencia, 46022 Valencia, Spain*
[b]*Departamento de Comunicaciones, Universitat Politecnica de Valencia, 46022 Valencia, Spain*
[c]*Departamento de Sistemas Informaticos y Computacion, Universitat Politecnica de Valencia, 46022 Valencia, Spain*

Abstract

Multichannel acoustic signal processing has undergone major development in recent years. The incorporation of spatial information into an immersive audiovisual virtual environment or into video games provides better sense of "presence" to applications. Spatial sound consists in reproducing audio signals with spatial cues (spatial information embedded in the sound) through headphones. This spatial information allows the listener to identify the virtual positions of the sources corresponding to different sounds. Headphone-based spatial sound is obtained by filtering different sound sources through special filters called *Head Related Transfer Functions* (HRTFs) prior to render them through headphones. Efficient computation plays an important role when the number of sources to be managed is high. This situation increases the number of filtering operations, requiring high computing capacity specially when the virtual sources are moving. Graphics Processing Units (GPUs) are high parallel programmable co-processors that provide massive computation when the needed operations are properly parallelized. This paper discusses the design, the implementation and the performance of a headphone-based spatial audio application whose processing is totally carried out on a GPU. This application is able to interact with the listener who can select and change the location of the sound sources in real-time. This work analyzes also specific computational aspects inside the CUDA environment in order to successfully exploit GPU resources. Results show that the proposed application is able to move up to 2500 sources simultaneously, while leaving free CPU resources for other tasks. This work emphasizes the importance of analyzing all CUDA aspects, since they can influence drastically the performance.

Keywords: GPU, CUDA, FFT, convolution, spatial sound, HRTF

1. Introduction

The growing need to incorporate new sound effects and to improve the hearing experience have increased the development of multichannel acoustic applications [1]. A spatial audio system based on headphones allows a listener to perceive the virtual position of a sound source [2]. These kind of effects are obtained by filtering sound samples with special filters whose coefficients shape the sound with spatial information. These filters are known as head-related transfer functions (HRTFs). The response of HRTFs filters describes how sound wave is filtered by properties of the

Email addresses: jobelrod@iteam.upv.es (Jose A. Belloch), mferrer@dcom.upv.es (Miguel Ferrer), agonzal@dcom.upv.es (Alberto Gonzalez), fjmartin@dcom.upv.es (F. J. Martinez-Zaldivar), avidal@dsic.upv.es (Antonio M. Vidal)
[1]Corresponding author

doi:10.1016/j.procs.2012.04.013

individual body shape (i.e., pinna, head, shoulders, neck, and torso) before the sound reaches the listeners eardrum [3]. When the number of sound sources increases, the achieved audio effects provide more realism to the scene. These spatial sounds are usually added to video games, video conferencing, movies, music performances, etc. However, when the number of sounds increases, the number of filtering increases, the quantity of processing executed in real-time increases and therefore, it is required high computational capacity. If a CPU processor were used to calculate the different filtering, the CPU processor would be overfull and the whole application would slow down. When this happens spatial sound information is usually avoided and unfortunately not added to the applications. This resources problem can be solved if these computational tasks are carried out by Graphics Processing Units (GPUs).

The appearance of CUDA [4] has allowed nowadays to use the GPU for applications beyond image processing. GPUs are high parallel programmable co-processors that provide efficient computation when the needed operations are properly parallelized. Programming efficiently a GPU requieres the knowledge of the architecture and how GPUs distribute their tasks among their processing units.

Hence, the objectives for this work focus on designing and implementing efficiently a headphone-based spatial sound application whose processing is totally carried out on a GPU. The application will be able to interact with the listener who will select, change and move the sound sources in real-time. In order to exploit efficiently GPU resources, the behavior of specific computational aspects inside the CUDA environment will have to be also studied. Once the application is efficiently implemented, its performance in terms of maximum number of sources in a real time application will be evaluated. The most valuable feature of the implementation is its scalability. The application must work in any GPU device, even a notebook GPU. Therefore, as it will be appreciated in the paper, the experiments are carried out in a GPU that belongs to a notebook.

Next section describes the meaning of head-related transfer functions HRTFs. Section 3 offers some details of the architectural and programming characteristics of the GPU, and reviews the state of art in which audio and GPU are involved. Section 4 describes the spatial sound application and how it interacts with the user. Implementation and evaluation of the GPU aspects inside CUDA environment is presented in Section 5, including global application performance. Finally, section 6 is dedicated some concluding remarks.

2. Head Related Transfer functions (HRTFs)

Individualized HRTFs are traditionally obtained either through measurements and extrapolations [5] or numerical simulation [6]. There exists a big quantity of HRTFs data-bases on internet. In these websites, a collection of HRTFs filters can be downloaded. Each pair of filters adds to the audio wave spatial information. When we listen the sound after filtering through headphones, we are able to identify the sound source in azimuth, between 0° and 360°, and in elevation, between -90° and +90°. Thus, a sound that comes from our left would correspond to position (0°, +90°), 0° in elevation and +90° in azimuth.

A data-base of HRTFs offers M different positions in the space, and therefore $2 \cdot M$ filters, since there is a filter for each ear. Figure 1 shows different positions in the space with their corresponding a pair of filters, the one for the left ear $h_l(\theta, \phi, n)$ and the other one for the right ear $h_r(\theta, \phi, n)$, where $\theta \in$[-90°, +90°], $\phi \in$[0°, +360°] and N represents the number of the coefficients of the filter. In total, there are $2 \cdot M$ filters of N coefficients. This means a database of $2 \cdot M$ vectors of N components.

We define $x(n)$ as a sound source. From standpoint of audio, $x(n)$ is a buffer composed of N audio samples where n is the index of the sample in the buffer. In a real-time audio application, when the buffer is fulled, it is sent to processing while another buffer is getting filled, and so on. Mathematically, $x(n)$ is an "input" vector of N components. Output buffers $y_r(n)$ and $y_l(n)$ represent the processed sounds that go to the right and the left headphones respectively (two "output" vectors of N components). They are obtained by convolving the sound source $x(n)$ with $h_r(\theta, \phi, n)$ and $h_l(\theta, \phi, n)$ respectively.

Therefore, to obtain output processed sounds from multiples sound sources, it will be necessary to add all the outputs before to render them. As an example, if we have two sources ($x_1(n)$ and $x_2(n)$) and we want $x_1(n)$ to be located in position (0°,90°) and $x_2(n)$ in position (225°,0°), the output sounds will be $y_l(n)$ and $y_r(n)$ for the left and right headphones respectively, as shown in Equation 1 where * denotes the convolution operation.

$$y_l(n) = h_l(0°, +90°, N) * x_1(n) + h_l(225°, 0°, N) * x_2(n), \quad (1)$$

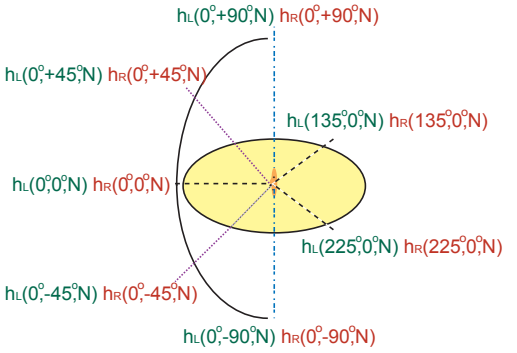

Figure 1: Two filters (left ear and right ear) for several positions in the space.

$$y_r(n) = h_r(0^\circ, +90^\circ, N) * x_1(n) + h_r(225^\circ, 0^\circ, N) * x_2(n).$$

2.1. Interaction with Listener

The problem occurs when sources moves, because for example a user has changed it manually using an interface or maybe applications, like a video game, require it. For example, source $x_1(n)$ moves from (0°,90°) to (0°,45°) and source $x_2(n)$ moves from (225°,0°) to (135°,0°). A common way to change the direction would be switching the filters. However, this switching can produce an audible clipping effect due to the abruptness of the changes. In [7], it is suggested to carry out a fade (a gradual increase or decrease in the level of an audio signal) with the current buffer in the moment of change. To this end, the operations that will be carry out with the input buffers ($x_1(n)$ and $x_2(n)$) are shown in Equation 2 where the new positions will be multiplied by a function $f(n)$ and the old ones by a function $g(n)$.

$$\begin{aligned} y_l(n) &= (h_l(0^\circ, +90^\circ, N) * x_1(n)) \cdot g(n) + (h_l(0^\circ, 45^\circ, N) * x_1(n)) \cdot f(n) + (h_l(225^\circ, 0^\circ, N) * x_2(n)) \cdot g(n) + (h_l(135^\circ, 0^\circ, N) * x_2(n)) \cdot f(n), \\ y_r(n) &= (h_r(0^\circ, +90^\circ, N) * x_1(n)) \cdot g(n) + (h_r(0^\circ, 45^\circ, N) * x_1(n)) \cdot f(n) + (h_r(225^\circ, 0^\circ, N) * x_2(n)) \cdot g(n) + (h_r(135^\circ, 0^\circ, N) * x_2(n)) \cdot f(n). \end{aligned} \quad (2)$$

Both functions $f(n)$ and $g(n)$ are vectors of N components that weight the results of the convolutions. The level of the output signal must not change. Figure 2 shows one of the summands to obtain $y_l(n)$ when $f(n)$ and $g(n)$ are *ramp functions* with different slope.

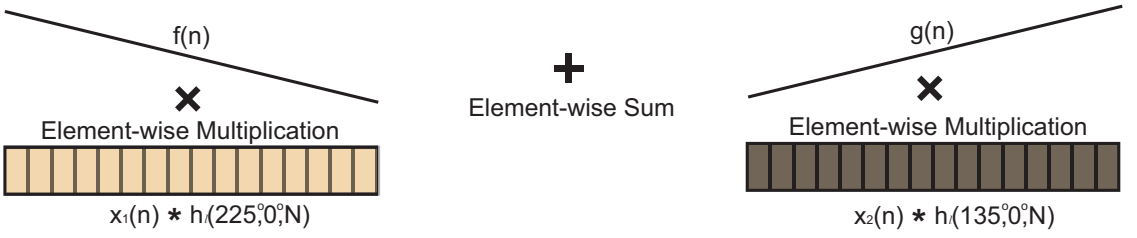

Figure 2: Change of virtual position of a sound signal by lineal fading. It shows one of the summands of Equation 2.

Functions $f(n)$ and $g(n)$ can exhibit different shapes as: *Sines and cosines*, *square roots*, *Hamming windows*, etc. A deep analysis to assess the use of a given shape is outside of the scope of this article. However, there has been carried out some subjective tests and it was realized that using *ramp functions*, the perception of spatial sound is really good and allows to identify the different sound sources. The actual implementation is totally independent of the values of $f(n)$ and $g(n)$. Therefore, when the positions change, additional processing is carried out.

3. GPU Graphics Processing Units

Compute Unified Device Architecture [4] is a software programming model that presents the massive computation potential offered by the programmable GPU. GPUs can have multiple stream multiprocessors (SM), where each stream multiprocessor consists of either eight cores if CUDA capability is 1.x, or 32 cores in the case of 2.x (GPU with Fermi architecture [8]). GPU devices may have a large amount of off-chip device memory (*global-memory*) and have a fast on-chip memory (*shared-memory*). In the CUDA model, the programmer defines the kernel function. The code that will be executed on GPU is written in the kernel.

A grid configuration, which defines the number of threads and how they are distributed and grouped, must be built into the main code. We can define a grid to be a mesh of blocks, each of them has a mesh of threads. Thus, a thread identification will be defined by a position within a block (*ThreadIdx.x*, *ThreadIdx.y*, and *ThreadIdx.z*), and this block will be defined within a grid (*BlockIdx.x*, *BlockIdx.y*, and *BlockIdx.z*). Parameters *BlockDim.x*, *BlockDim.y*, and *BlockDim.z* indicate the dimensions of blocks in the same way as *gridDim.x*, *gridDim.y*, and *gridDim.z* indicate the dimensions of grid.

At runtime, the kernel distributes all the thread blocks among SMs. Each SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps that get scheduled by a *warp scheduler* for execution. The way a block is partitioned into warps is always the same; each warp contains consecutive threads, increasing thread `Idx` with the first warp containing thread `Idx=0`. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path, it means, there is no warp serialization. It is important to have multiple warps in a block due to it allows to hide latency. It means, if all threads of a warp must carry out a memory access, this access can take several clock cycles. In order to hide latency, the *warp scheduler* selects a warp that is ready to execute. The *warp scheduler* switchs among different warps in order to try full utilization of SMs.

An important aspect of accessing to *global-memory* is to do it in a Coalesced way. Coalescing means that the threads are writing into a small range of memory addresses with certain patern. For example, considering an `array` pointer to *global-memory* and `idx` as the identification of a thread, if thread `idx` writes to address `array[idx]` and thread `idx+1` to address `array[idx+1]`, we achieve good coalescing.

3.1. State of Art

The use of GPUs [9] has always been related to graphic or image applications. Since the appearance of CUDA programming, there is a big field of researches that have already carried out seeking better performances. This is the case of different computational cores, such as matrix multiplication [10], Boltzmann equation [11], or Parallel 3D fast wavelet transform [12].

GPU computing has already been applied to different problems in Acoustics and Audio processing. Studies of computing room acoustics were carried out by Webb and Bilbao in [13] and [14], as well as geometric acoustic modelling like ray-tracing [15] [16]. Spatial sound has also made use of GPUs [17].

Focusing on the convolution operation, there are some publications in the literature that implement the convolution on GPU. Cowan and Kapralos were apparently the first ones to implement a convolution algorithm on GPU using the OpenGL shading language [18] in [19]. The work shown in [20] presents multiple convolutions carried out concurrently.

4. Headphone-based Spatial Sound application

As it was described in Section 1, the key of this kind of applications is to convolve each data buffer source with its HRTF filters (for the left and right headphone respectively) corresponding to the direction where the source is virtually located. At the beginning, all the HRTF filters are transferred from CPU to GPU. Then, FFT transformations are carried out over all the filters to get their frequency responses. The steps of the application will be:

1. Buffer filling of audio samples from different sound sources. Mathematically, obtain multiple vectors $x_z(n)$, where z indicates the source number. We will call *NumSources* to the number of sources in the application. Thus, there will be as much $x_z(n)$ as number of sources, $z \in [0, NumSources - 1]$.
2. Transfer *NumSources* vectors $x_z(n)$ to GPU.

3. Transfer a position vectors called *POS*. The length of *POS* is *NumSources*. However, the corresponding vector in the GPU *gPOS* will be $2 \cdot NumSources$. The second half of *gPOS* is used to store the new positions when the sources are changing. When this occurs, double processing must be executed. See subsection 2.1 and 4.1
4. The convolution operation is carried out as it is reported in [20]:
   (a) Make a FFT transformation of each vector $x_z(n)$.
   (b) Make element-wise multiplications of each $x_z(n)$ with its corresponding filters $h_r(z,n)$ and $h_l(z,n)$. These element-wise multiplications are possible due to all vectors will be in the frequency domain [21].
   (c) Make an element-wise sum of all the outputs, as shown in Equation 1.
   (d) Make two inverse FFT transformations of each output vectors from frequency domain to time domain $y_l(n)$ and $y_r(n)$.
   (e) In case of directions changing, outputs vectors will be weighted by the functions $f(n)$ and $g(n)$. See section 4.1
5. Transfer vectors $y_l(n)$ and $y_r(n)$ from GPU to CPU. Then, samples are ready to be reproduced through headphones.

In a real-time audio application, the step 1) is executed during the whole application, due to, each input audio sample from each source arrives to the buffer with a standard sample period of (1/44.1) ms. (CD sound quality has a sample frequency of 44.1 kHz). The application works with two audio buffers: A-buffer and B-buffer. When A-buffer is fulled, this is transfered to GPU and the processing begins. Meanwhile, the audio samples fills the B-buffer. In order to achieve best performance of the application, the processing of A-buffer must end (steps 2,3 and 4) before the B-buffer gets filled (step 1). Afterwards, a switching is produced, B-buffer is transfer to GPU to begin processing while A-buffer gets filled. The applications ends when there are no more audio samples to process.

The time to fill the buffer t_{buff} is calculated as $N/44.1$ and it is independent of the number of sources. In contrast, the processing time t_{proc} depends on the number of sources *NumSources* (increasing *NumSources*, the number of operations to execute increases) and on the use of the GPU resources to carry out the processing. Therefore, it is important to develop an efficient implementation on GPU that allows to achieve maximum *NumSources* that satisfy $t_{\text{proc}} < t_{\text{buff}}$.

4.1. Interaction with user

In order to interact with the user, this application uses the *stream* object. CUDA applications manage concurrency through *streams*. A *stream* is a sequence of commands that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently. *Streams* object is used mainly to overlap computation with transfer data CPU $\Leftrightarrow$ GPU. In this sense, the application is running in the stream 0 while stream 1 works as a *listener*, that hears the changes of the position. Once it is detected the change, stream 1, transfers new positions to GPU. As this transfer is carried out in a different stream, we are allowing that this transfer could be overlapped with the kernel processing. The application detects the arrival of new directions and with the next buffer of samples, executes the operations of Equation 2. To this end, the modifications of steps 4.b and 4.c are as easy as to simulate, only for this buffer of samples, that value of *NumSources* is $2 \cdot NumSources$ and the number of outputs is 4 (2 for the left earphone and 2 for the right earphone). After step 4.d, the 4 outputs are reduced to 2 outputs, by weighting the outputs with functions $f(n)$ and $g(n)$, as it is indicated in Section 2.1. Finally, the two outputs are sent back to CPU, as usual.

5. Implementation and Performance

From the previous steps, steps 4.a and 4.d are carried out automatically by NVIDIA FFT library, CUFFT [4]. It is important to design efficient kernels for element-wise multiplications (step 4.b) and element-wise sum (step 4.c). The audio application is executed on a computer whose GPU characteristics are shown in Table 1.

Furthermore, there are some advices in [4] in order to obtain best performance in a CUDA application: maximum coalescence in access to *global-memory* and minimum warp serialization; to hide latency arising from register dependencies, maintain approximately 25% occupancy as a minimum; and the number of threads per block should be a multiple of warp size.

CUDA Device	GTS-360M
CUDA Architecture	TESLA
CUDA Capability	1.2
Number of SMs	12
Maximum number of active blocks per SMs	8
Maximum number of active threads per SMs	1024
Maximum number the threads per block	512
Maximum number of registers per SMs	15689
Maximum number of active warp per SMs	32
Overlap Transfer with computation	One direction: CPU⇒GPU or CPU⇐GPU

Table 1: Characteristics of the GPU.

5.1. Element-wise multiplication

This kernel devotes a GPU thread to: take one component from one of the $x_z(n)$, multiply this component by its correspondent component of $h_l(\theta,\phi,N)$ or $h_r(\theta,\phi,N)$ and store it in a *Result Matrix* whose dimensions will be (2· *NumSources* × *N*). Therefore, this kernel uses 2· *NumSources* · *N* threads.

Once it is clear the task of every thread, it is time to launch the kernel. To this end, it is necessary to organize them, as it is indicated in Section 3, in blocks of threads and configure a CUDA grid of blocks. Figure 3 shows the operations that carry out a thread, showing also the different blocks of the grid.

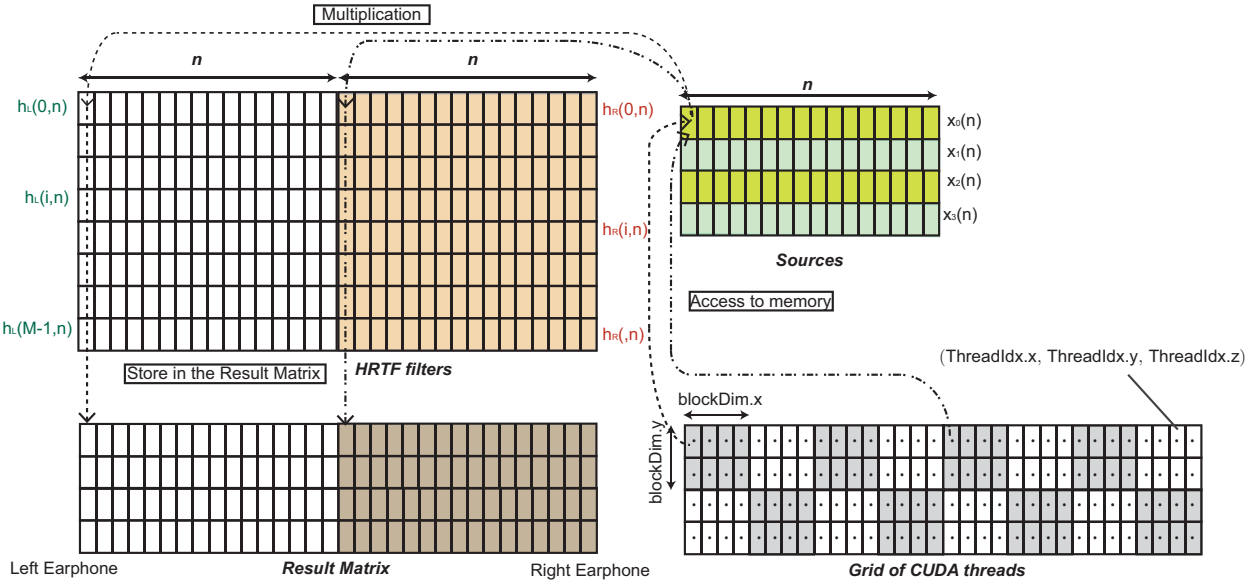

Figure 3: Operations that carry out a cuda thread inside its block. Each point represent a thread.

5.1.1. Performances

Following the previous points, we set *blockDim.x* to 32 and *blockDim.y* to *NumSources*. In case the number of sources is greather than 16, we should set *blockDim.y*=16 and therefore, to have more than one *gridDim.y*; *gridDim.y*= *NumSources*/*blockDim.y*. Also, *gridDim.x*= 2·*n*/*blockDim.y*. Table 2 shows different tests with different parameters from Visual Profiler (a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++).

As it is observed in Table 2, maximum occupancy does not mean best performance, however, minimum occupancy means worse performance. There must be a balance among distribution of threads in blocks, the number of blocks,

Num TEST	Num Sourc.	GPU time	Grid size	Block size	Reg. Ratio	Ac.Blc. per SM	Ac.Thr. per SM	Occu (Ac.W.)	Occu L.Fact	gld & gst 32	gld & gst 64	gld & gst 128	Warp Ser.
1	1	5.92 us	64 x 1	32 x1	0.5	8:8	256:1024	8:32	block-size	30	0	160	0
2	**1**	**5.32** us	**32** x **1**	**64** x **1**	**0.5**	**8:8**	**512:1024**	**16:32**	**block-Size**	**28**	**0**	**92**	**0**
3	1	6.24 us	16 x1	128 x1	1	8:8	1024:1024	32:32	None	32	0	96	0
4	1	6.42 us	8 x 1	256 x1	0.875	4:8	1024:1024	32:32	None	32	0	96	0
5	1	7.42 us	4 x 1	512 x 1	0.656	3:8	384 : 1024	12:32	Reg.	32	0	98	0
4	**128**	**206.3** us	**64** x **16**	**32** x **8**	**0.875**	**4:8**	**1024:1024**	**32:32**	**None**	**4096**	**0**	**12369**	**0**
3	128	212.3 us	64 x 8	32 x 16	0.812	2:8	1024:1024	32:32	None	4000	0	12448	0
5	**256**	**430.0** us	**64** x **32**	**32** x **8**	**0.875**	**4:8**	**1024:1024**	**32:32**	**None**	**8192**	**0**	**24656**	**0**
6	256	444.8 us	64 x 16	32 x 16	0.812	2:8	1024:1024	32:32	None	8192	0	24640	0

Table 2: Parameters of Kernel 1 obtained from Visual Profiler except GPU time which is obtained through the Time measures. The columns indicate from left to right: Identification of the test, Number of Sources used for the test, time employed by GPU, grid size, block size, registers ratio, active blocks per SM, active threads per SM, Occupancy (Active warps per SM), Limiting factor of the occupancy, number of access to global memory in blocks of 32 bytes both in load data and store data, as previous one but in blocks of 64, as previous one but in blocks of 128, number of times that a warp serialization is produced.

etc. From the tests carried out with one source, the most efficient implementation is the one that corresponds to test 2 because there is middle occupancy, and there exists more coalescence since there is less access to *global-memory* both 32 and 128 accesses. Also there is no warp serialization. Therefore, it is understandable that the test 2 is the one which exhibits the smallest time. Regarding implementations with multiples sources, it is observed that implementations with blocksizes 32 x 8 achieve better performances. They have less accesses to memories of 32 and 128 and the time is small. Coalescing access was suspected a priori, since the implementation, showed in Figure 3, followed the programming guidelines enumerated in Section 3.

5.2. Element-wise sum

Next step is to sum all the outputs components, it means, to carry out a reduction of all the rows. NVIDIA recommends a reduction algorithm in [22]. This algorithm takes advantage of the *shared-memory*. It consists in that threads, inside the same block bring firstly to *shared-memory* 2·*blockDim.x* components. Then, in the *shared-memory*, each thread executes a sum between two components. The previous step is repeated until all components are added. So that, for each time, a sum is executed, it is used half threads respect the previous iteration. To adapt this reduction algorithm to our data structure, we must apply it to every column. Therefore, unidimensional blocks will be defined, once per column. Figure 4 shows the adaptation of this algorithm to our data structure.

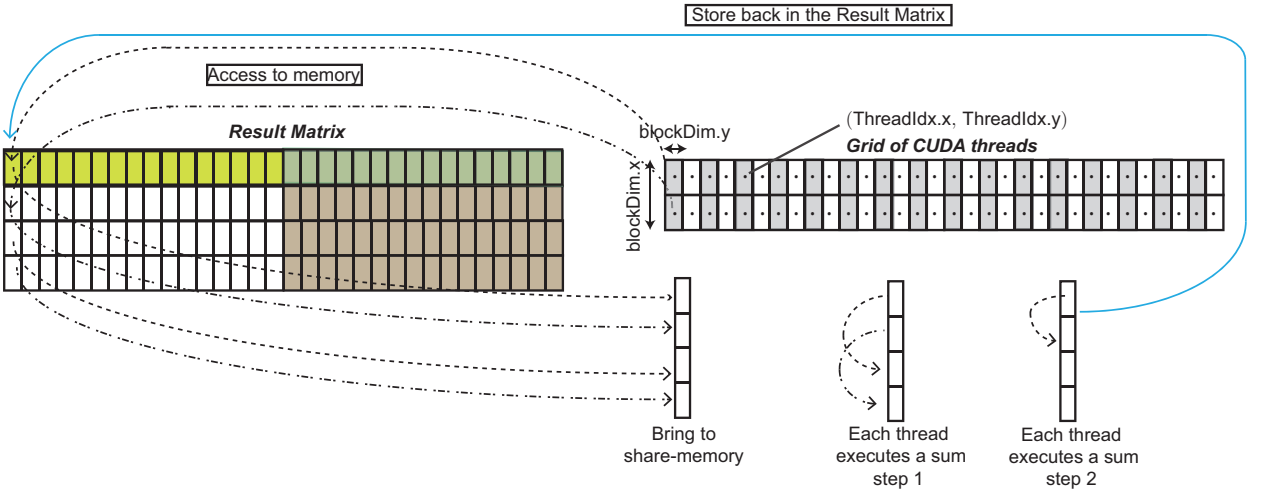

Figure 4: This kernel makes a reduction of all the rows using *shared-memory*.

The problem exists when there is more than 1024 sources. As maximum blocksize is 512 threads, more blocks are needed. When this happens, each block makes a reduction of 1024 elements. However, in order to complete the

reduction process, another kernel must be launched, due to blocks must exchange values. Moreover, coalescing access is not warranted due to each thread inside the same block is $2 \cdot N$ memory positions separated.

Therefore, seeing the limitations of the previous implementation, it was decided to carry out an alterntive kernel which will be totally coalescing and also independent of the number of sources. This kernel will have unidimensional blocks and will be configured by $2 \cdot N$ threads in total. Each thread will carry out *NumSources* sums. Figure 5 shows how this kernel works.

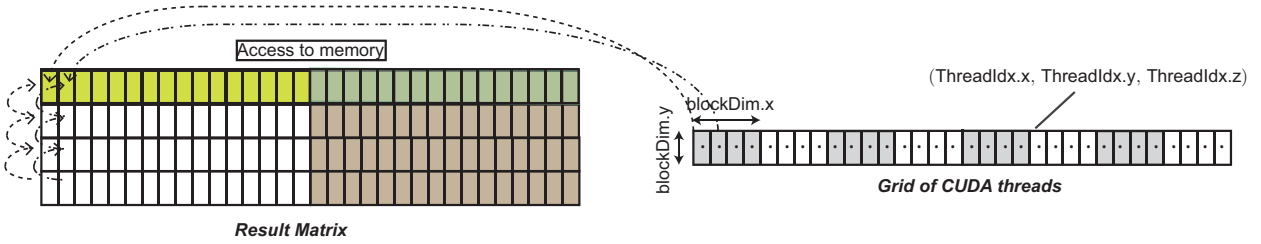

Figure 5: This kernel makes a reduction of all the rows. Each thread executes *NumSources* sums.

5.2.1. Performance

Once it is described the two kinds of kernels, it is important to compare the two implementations. Figure 6 shows the difference performance between the two previous implementations, the one using *shared-memory* (kernel 2A) against the other one that favors coalescing (kernel 2B).

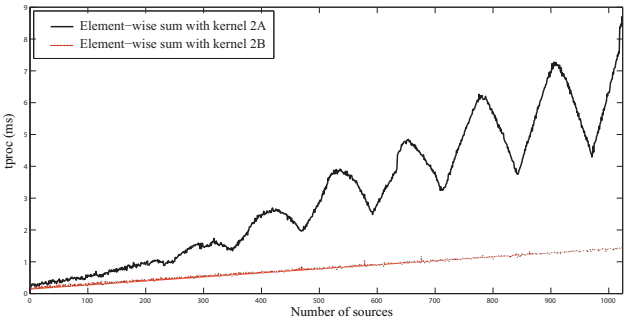

Figure 6: Comparation of times used for kernels 2A and 2B as the number of sources increases.

The maxima and minima in the kernel 2A curve are produced by the warp serialization (Figure 6). This means that it exists bank-memory conflicts in shared-memory. In Table 3, it is shown, as an example, the performance with 710, 780 and 840 sources obtained from Visual Profiler.

Num TEST	Num Sourc.	GPU time	Grid size	Block size	Reg. Ratio	Ac.Blc. per SM	Ac.Thr. per SM	Occu (Ac.W.)	Occu L.Fact	Warp Ser.
1	710	3.11 ms	1 x 2048	355 x1	0.37	2:8	710:1024	8:32	block-size	454407
2	780	5.98 ms	1 x 2048	390 x1	0.43	2:8	780:1024	26:32	block-size	579378
3	840	3.63 ms	1 x 2048	420 x1	0.43	2:8	840:1024	28:32	block-size	569362

Table 3: Parameters of Kernel 2A obtained from Visual Profiler except GPU time which is obtained through the Time measures.The columns indicate from left to right: Identification of the test, Number of Sources used for the test, time employed by GPU, grid size, block size, registers ratio, active blocks per SM, active threads per SM, Occupancy (Active warps per SM), Limiting factor of the occupancy,number of times that a warp serialization is produced.

Therefore, for the application, it will be used the second implementation corresponding to kernel 2B due to it achieves best performances. Table 4 shows the performance parameters of kernel 2B.

Num TEST	Num Sourc.	GPU time	Grid size	Block size	Reg. Ratio	Ac.Blc. per SM	Ac.Thr. per SM	Occu (Ac.W.)	Occu L.Fact	gld & gst 32	gld & gst 64	gld & gst 128	Warp Ser.
1	8	13.79 us	64 x 1	32 x1	0.25	8:8	256:1024	8:32	block-size	0	0	704	0
2	8	13.21 us	32 x 1	64 x1	0.25	8:8	512:1024	16:32	None	0	0	704	0
3	**8**	**13.08** us	**16** x**1**	**128** x**1**	**0.5**	**8:8**	**1024:1024**	**32:32**	**None**	**0**	**0**	**704**	**0**
4	8	13.85 us	8 x 1	256 x 1	0.5	4:8	1024:1024	32:32	None	0	0	704	0
5	8	13.91 us	4 x 1	512 x 1	0.5	2:8	1024:1024	32:32	None	0	0	704	0

Table 4: Parameters of Kernel 2B obtained from Visual Profiler except GPU time which is obtained through the Time measures. The columns indicate the same as the ones in Table 2.

As it is observed in Table 4, in kernel 2B, occupancy plays an important role, since the configuration with best performance (Test num. 3) gets maximum occupation in threads and blocks actives per SMs. In this case, coalesced access is easily achieved in every configuration.

5.3. Application Performance

Once it is set the configuration parameters, it is moment of analyzing the capacity of the spatial application in real-time. To this end, we measure t_{proc} (see Section 4) increasing the number of sound sources in the application. The HRTF data base used for the experiments can be found in [23] and it is composed of 384 filters (187 filters for left earphone and 187 filters for the right earphone) of 512 components, therefore M=187 and N=512. Thus, time t_{buff} is 11.61 ms. Therefore, all configurations whose time t_{proc} is greater than the time t_{buff} will not work in a real time application. The experiments have been carried out in the extreme worse situation, that is, when all the sources are moving at the same time. Figure 7 shows time t_{proc} for different configurations. It is also marked the time t_{buff} which symbolizes the border between off-line applications and real-time applications. Obviously, if this application has to share GPU resources with other applications, peak performance would decrease.

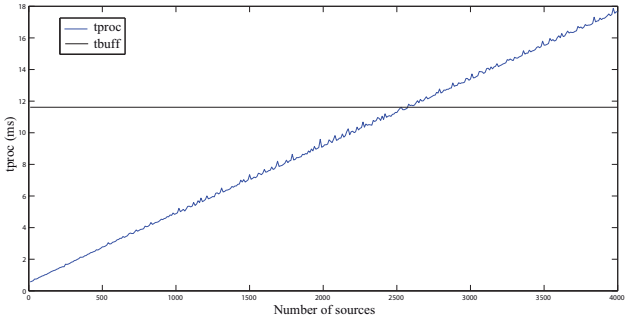

Figure 7: Number of sources that can be managed by a spatial sound application on GPU in real time.

6. Conclusions

The work we present focuses one of the problems of audio engineering: to add spatial information to multichannel sound applications. A common GPU with CUDA capabilities has been used. This work has demonstrated that if the computational load, that the audio processing generates, is carried out by a GPU, it is possible to manage applications with multiples sources and not to overload the CPU.

In order to develop efficiently an implementation of the spatial sound on GPU, it was necessary to analyze specific computational aspects inside the CUDA environment in order to successfully exploit GPU resources. Therefore, each kernel has been watched closely. Aspects such as occupancy, coalescing, warp serialization has been taken into account in order to achieve best performance. This study has allowed us to set CUDA parameters whose values determinate the global application performance.

Once the CUDA parameters has been set, and the application works perfectly, it has been carried out the application to the extreme conditions in which a user changes simultaneously multiple sources, increasing the quantity of

operations in order to see the limits of the application when a real time conditions exist. The results show that this application is able to manage up to 2500 sources simultaneously, and the task manager of the computer reaches barely 10%.

As a final conclusion, we must point out two important aspects. The first one is related to the work methodology. It is important to carry out an analysis of cuda aspects, before configuring a grid of CUDA threads, due to the distribution of the cuda threads inside the blocks and these ones inside the grid is totally crucial for performances, as Table 2 and Table 4 show. Finally, with the execution of the audio processing on the GPU, we have substantially reduced the computational load of the CPU, which means that CPU resources can be used for other tasks while GPU performs an audio application of more than 1000 sources.

Acknowledgements

This work has been partially funded by Spanish Ministerio de Ciencia e Innovacion TEC2009-13741, Generalitat Valenciana PROMETEO 2009/2013 and GV/2010/027, and Universitat Politècnica de València through Programa de Apoyo a la Investigación y Desarrollo (PAID-05-11).

References

[1] E. Torick, Highlights in the history of multichannel sound, J. Audio. Eng. Soc. 46 (5) (1998) 27–31.
[2] V. Algazi, R. Duda, Headphone-based spatial sound, IEEE Signal Processing Magazine 28 (1) (2011) 33–42.
[3] J. Blauert, Spatial Hearing - Revised Edition: The Psychophysics of Human Sound Localization, The MIT Press, 1996.
[4] NVIDIA Documentation, online at: http://developer.download.nvidia.com/.
[5] S. Spors, J. Ahrens, Efficient range extrapolation of head-related impulse responses by wave field synthesis techniques, in: IEEE International Conference on Acoustics, Speech and Signal Processing, Prague, Czech Republic, 2011.
[6] Y. Kahana, P. A. Nelson, Numerical modelling of the spatial acoustic response of the human pinna, Journal of Sound and Vibration 292 (1-2) (2006) 148 – 178. doi:10.1016/j.jsv.2005.07.048.
[7] A. Kudo, H. Hokari, S. Shimada, A study on switching of the transfer functions focusing on sound quality, Acoustical Science and Technology 26 (3) (2005) 267–278.
[8] NVIDIA Next Generation: FERMI, online at: http://www.nvidia.com/.
[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (10) (2008) 1370–1380.
[10] K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, S. G. Sedukhin, Multi-level optimization of matrix multiplication for GPU-equipped Systems, Procedia Computer Science 4 (0) (2011) 342 – 351, proceedings of the International Conference on Computational Science, ICCS 2011.
[11] Y. Kloss, P. Shuvalov, F. Tcheremissine, Solving Boltzmann equation on GPU, Procedia Computer Science 1 (1) (2010) 1083 – 1091, proceedings of the International Conference on Computational Science, ICCS 2010.
[12] J. Franco, G. Bernab, J. Fernndez, M. Ujaldn, Parallel 3d fast wavelet transform on manycore GPUs and multicore CPUs, Procedia Computer Science 1 (1) (2010) 1101 – 1110, proceedings of the International Conference on Computational Science, ICCS 2010.
[13] C. J. Webb, S. Bilbao, Virtual room acoustics: A comparison of techniques for computing 3D-FDTD schemes using CUDA, in: Proceedings of the 130th AES Convention, London, U.K., 2011.
[14] C. J. Webb, S. Bilbao, Computing room acoustics with CUDA - 3D FDTD schemes with boundary losses and viscosity, in: IEEE International Conference on Acoustics, Speech and Signal Processing, Prague, Czech Republic, 2011.
[15] M. Jedrzejewski, K. Marasek, Computation of room acoustics using programmable video hardware, Computational Imaging and Vision 32 (2006) 587–592, http://dx.doi.org/10.1007/1-4020-4179-9_84.
[16] N. Rober, U. Kaminski, M. Masuch, Ray acoustics using computer graphics technology, in: Conference on Digital Audio Effects (DAFx-07) proceedings, Bourdeaux, France, 2007.
[17] B. Cowan, B. Kapralos, Spatial sound for video games and virtual environments utilizing real-time GPU-Based Convolution, in: Proc. 2008 Conf. on Future Play: Research, Play, Share, Ontario, Canada, 2008.
[18] openGL, online at: http://www.opengl.org/.
[19] B. Cowan, B. Kapralos, GPU-Based One-Dimensional Convolution for Real-Time Spatial Sound Generation, Loading...: The Journal of the Canadian Game Studies Association 3 (5) (2009) 1–14.
[20] J. A. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, A. M. Vidal, Real-time massive convolution for audio applications on GPU, Journal of Supercomputing 58 (3) (2011) 449–457.
[21] A. V. Oppenheim, A. S. Willsky, S. Hamid, Signals and systems, 2nd Edition, Processing series, Prentice Hall, 1997.
[22] D. B. Kirk, W.-m. W. Hwu, Programming Massively Parallel Processors, Morgan Kaufman, 2010.
[23] Listen HRTF database, online at: http://recherche.ircam.fr/equipes/salles/listen/index.html.